Proving inductive equalities algorithms and implementation

Azzedine Lazrek, Pierre Lescanne, Jean-Jacques Thiel

INRIA

UNITE DE RECHERCHE
INRIA-LORRAINE

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 682

PROVING INDUCTIVE EQUALITIES ALGORITHMS AND IMPLEMENTATION

Azzedine LAZREK
Pierre LESCANNE
Jean-Jacques THIEL

JUIN 1987

PROVING INDUCTIVE EQUALITIES

ALGORITHMS AND IMPLEMENTATION

PREUVE D'EGALITE PAR RECURRENCE

ALGORITHMES ET IMPLANTATION

PAR

AZZEDINE LAZREK

PIERRE LESCANNE

JEAN-JACQUES THIEL

Résumé : Le but de cet article est, tout d'abord, de décrire un algorithme pour tester la complétude suffisante et ensuite de présenter les concepts nécessaires pour comprendre le comportement d'une implantation d'un démonstrateur automatique de théorèmes de propriétés par récurrence pour des programmes fonctionnels ou des spécifications de types abstraits de données. Ces programmes ou spécifications sont des systèmes de réécriture où des relations entre constructeurs sont autorisées.

La méthode est essentiellement fondée sur une preuve par consistance implanté au travers d'une complétion de Knuth-Bendise qui étend l'approche de Huet- Hullot sous plusieurs aspects.

Cecla suppose que l'on sait prouver la complétude inductive d'un ensemble de relations parmi des constructeurs et la complétude suffisante ou relative de définitions de fonctions. Après avoir introduit le concept de complétude inductive, des théories qui sont inductivement complètes sont présentées. D'autre part, un test de complétude relative est implanté par une extension d'un algorithme qui fonctionnait auparavant seulement dans le cas d'absence de telles relations.

Mots -clés : spécifications, modèle, algèbre libre, algèbre initiale, procédure de complétion, système de réécriture de termes, terminaison, paires critiques, confluence, récurrence.

Abstract : The aim of this paper is first to describe an algorithm for testing sufficient completeness and second to present concepts necessary to understand the behavior of an implémentation of an automatic prover of inductive properties of functional programs or specifications of abstract data types. These programs or specifications are rewriting systems and relations between constructors are allowed.

The method is essentially based on a proof by consistency implemented through a Knuth-Bendix completion, extending the Huet-Hullot approach in many respects. This requires to prove the inductive completeness of the set of relations among the constructors, and the relative (or sufficient) completeness of the definitions of the function. After introducing the concept of inductive completeness, inductively complete theories are presented. On the other hand a test of relative completeness is implemented by an extension of an algorithm that worked only when no relation among constructors existed.

Keyworlds : specification, model, free algebra, initial algebra, completion procedure, term rewriting systemn, termination, critical pairs, confluence.

Proving Inductive Equalities
Algorithms and Implementation

Azzeddine LAZREK, Pierre LESCANNE
Jean-Jacques THIEL☆

Centre de Recherche en Informatique de Nancy
Campus Scientifique BP 239
54506 Vandoeuvre-les-Nancy Cedex FRANCE

ABSTRACT: The aim of this paper is first to describe an algorithm for testing sufficient completeness and second to present concepts necessary to understand the behavior of an implementation of an automatic prover of inductive properties of functional programs or specifications of abstract data types. These programs or specifications are rewriting systems and relations between constructors are allowed.

The method is essentially based on a proof by consistency implemented through a Knuth-Bendix completion, extending the Huet-Hullot approach in many respects. This requires to prove the inductive completeness of the set of relations among the constructors, and the relative (or sufficient) completeness of the definitions of the function. After introducing the concept of inductive completeness, inductively complete theories are presented. On the other hand a test of relative completeness is implemented by an extension of an algorithm that worked only when no relation among constructors existed.

KEYWORDS: specification, model, free algebra, initial algebra, completion procedure, term rewriting system, termination, critical pairs, confluence.

☆*Current adress: CERILOR Rue André Fruchard 54320 Maxéville FRANCE.*

1 – INTRODUCTION

Rewriting systems provide a nice mechanism for functional programming language. Their basic feature is a *call by matching* and they are included in actual languages like HOPE ‹BMS,80›, ML ‹MIL,84›, ‹CPH,85›, MIRANDA ‹TUR,85›, OBJ2 ‹FGJM,84›. In this paper, our main concern is to prove properties of functions expressed as equalities and usually proved by induction, ‹GOG,80›, ‹H-H,82›, ‹MUS,80›, ‹LAN,81›, ‹REM,82›, ‹M-G,83›, ‹PAU,84›, ‹PUE,84›, ‹KIR,84›, ‹KOU,85›, ‹J-K,86›. For example, if we define a function *flatten* by:

[ ] @ x → x
x @ [ ] → x
(x @ y) @ z → x @ (y @ z)
flatten([ ]) → [ ]
flatten([x]) → *flatten*(x)
flatten(*a*) → *a*
flatten(*b*) → *b*
flatten(*a* @ x) → *a* @ *flatten*(x)
flatten(*b* @ x) → *b* @ *flatten*(x)
flatten([x] @ y) → *flatten*(x) @ *flatten*(y)

in the theory where the constructors are [ ], [_], *a*, *b* and @, we may want to prove that *flatten* is an involution in other words *flatten*(*flatten*(x)) = *flatten*(x) or *flatten* is a morphism for @ that is *flatten*(x @ y) = *flatten*(x) @ *flatten*(y). The method we present does not use explicitly an inference rule, but a proof by absence of contradiction ‹M-K,84›.

We do not introduce many new concepts, but we give a synthetic presentation of the theory illustrated by examples and we set the theoretical background of our implementation, in a language we wish as simple as possible. However the presentation of the algorithm for proving relative completeness ‹THI,84› i.e. the second part of the paper is new.

The paper is divided into two parts. In the first part we describe and prove the induction principle based on a completion procedure and an inductive completeness of the relations among constructors. This relies on a relative completeness procedure described and proved in the second part. A method for checking the completeness and examples run in an extension of REVE ‹LES,83›, ‹FOR,84›, ‹F-G,83› are presented in Appendix 3. This method improves the implementation proposed in ‹FAG,84›, ‹H-H, 82› in two respects. An algorithm checks the relative completeness. Relations among constructors are allowed.

2 – PROOF BY CONSISTENCY

2.1 – Notations

Let F be a set of operators or functions, X a set of variables. Suppose F divided into a set C of constructors, and a set D of defined operators. *Constructors* are supposed to describe each object of the type. *Defined operators* are functions defined on the abstract data type and are supposed to disappear in the computation of the values of the function on objects of the type. We suppose that C is not empty and X, C and D are disjoint. In addition, we suppose that there exists only one sort. Actually, this is not a restriction and the result of this paper can be extended to many-sorted data types.

Let us take the following notations ‹KIR,85›:

T(F,X) = the set of terms constructed using operators in F and variables in X.
T(F) = the set of ground terms on F, i.e., without variables.
T(C,X) = the set of terms built with only constructors in C.
T(C) = the set of ground terms built on C.
S = the set of sorts.
A = the set of axioms i.e., pair of terms.
(S,F,A) = a specification with sort S, operators F and axioms A.
M(A) = the class of models of (S,F,A), the class of F-algebras that satisfy A.
T(F,X,A) = the free algebra on X.
T(F,A) = the initial algebra of M(A).
In this paper $\oplus$ denotes the disjoint union.

2.2 – Definitions

Let u and v be two terms of T(F,X) and let us define the two congruences over T(F,X) as follows ‹HEN,77›:

Definition: *equational equality*
$u =_A v$ (or u = v is *an equational theorem* in the theory generated by A) iff u = v is valid in M(A), i.e., valid in every algebra that satisfies A. From the completeness theorem for equational deduction, this means that u = v is a consequence of A, usually denoted by A |-- u = v. This also means that u = v is valid in the free algebra T(F,X,A) sometimes denoted by T(F,X,A) |== u = v.

Definition: *inductive equality*
$u =_{ind(A)} v$ (or u = v is *an inductive theorem*) iff u = v is valid in T(F,A), written T(F,A) |== u = v. This is equivalent to $\sigma(u) =_A \sigma(v)$ for all ground substitution σ.

Generally, to prove the validity of an equation in T(F,A), we need an inductive reasoning, when an equational reasoning is sufficient in M(A).

Example: Peano natural numbers
Let C={ 0 ; S } , D={ + } and suppose the set of axioms is:

$0 + x \quad == x$

$S(x) + y == S(x + y) \quad (NAT)$

The following equation is an equational theorem of *NAT*

$(S(x) + y) + z = S((x + y) + z)$

and the following equations are inductive theorems that are not equational:

$x + 0 \quad = x$

$x + S(y) \quad = S(x + y)$

$x + y \quad = y + x$

$x + (y + z) = (x + y) + z$

In a structured specification of an abstract data type the set of axioms A is divided into two subsets, DA and CA, as follows:

DA = the specification of defined operators.

CA = the set of constructor relations, possibly empty.

From a functional programming point of view, CA defines the data structure one works on and DA is the definition of a set of functions i.e., a functional program.

Usually two important concepts are attached to structured algebraic specifications defined as follows:

Definition: *(relative) completeness*

The specification (S,F,A) is complete w.r.t. (S,C,CA) iff for every $u \in T(F)$ there is a $v \in T(C)$ s.t. $u =_A v$.

Definition: *(relative) consistency*

The specification (S,F,A) is consistent w.r.t. (S,C,CA) iff for every u and v in T(C) $u =_A v \Longrightarrow u =_{CA} v$.

This means that if (S, F, A) is a specification relatively complete and consistent w.r.t. (S, C, CA) then $T(C)/=_{CA}$ is isomorphic to $T(F)/=_A$, i.e., the algebra on the classes of ground terms modulo CA is the algebra on the classes of ground terms modulo A.

The relative completeness is called sufficient completeness by J.V. Guttag and J.J. Horning ‹G–H,78› or principle of definition by G. Huet and J.M. Hullot ‹H–H,82›.

Often one would like to be able to "decide" i.e., to prove or disprove, inductive theorems. As it will be seen later on, this is especially necessary in the theory (S,C,CA) of the relations among the constructors. This will work if the inductive

theory coincides with the equational theory and the latter is decidable, for instance, is associated with a convergent term rewriting system. In other words, we require that all inductive consequences of the set of relations between constructors are obtained by equational reasoning. This is called the *inductive completeness* by E. Paul ‹PAU,84›. This concept was also presented by J. Heering under the name of "ω–completeness" ‹HEE,85›, by L. Henkin ‹HEN,77› from a logician viewpoint, by W. Taylor ‹TAY,79› from a universal algebra viewpoint and by G.D. Plotkin ‹PLO,74› who shows that $\lambda\kappa\beta\eta$–calculus is not ω–complete.

Definition: *inductive completeness*
The specification (S,F,A) is inductively complete iff every inductive theorem is an equational theorem.

Examples:
The following specifications (S,F,A) are inductively complete:
F = { 0 , succ , pred } ; A = { succ(pred(x)) = x , pred(succ(x)) = x },
easy using the method shown in Appendix 2.
F = { 0 , 1 , + } ; A = { 0+x = x , x+y = y+x , x+(y+z) = (x+y)+z } ‹PAU,84›.
F = { 0 , opp , succ } ; A = { opp(0)=0 , opp(opp(x))=x , succ(opp(succ(x)))=opp(x) }
‹Appendix 2›.
F = { 0 , a , b , . } ; A = { 0.x = x , x.0 = x , x.(y.z) = (x.y).z } ‹Appendix 2›.
F = { 0 , [_] , @ } ; A = { 0 @ x = x , x @ 0 = x , x @ (y @ z) = (x @ y) @ z }
generalization of the previous one.
F = {0, 1, +, opp}; A = {0+x = x, x+y = y+x, x+(y+z) = (x+y)+z, opp(0) = 0,
opp(opp(x)) = x, opp(x+y) = opp(x) + opp(y)} ‹Appendix 2›.
The following specification (S,A,F) is not inductively complete:
F = { 0 , 1 , + } ; A = { 0+x = x , x+0 = x , x+(y+z) = (x+y)+z }
because T(F,A) |== x+y = y+x.

Since the inductive completeness is undecidable, we assume it is tested by the user.

2.3 – The intuitive idea behind the proof by consistency

Completeness induces a very nice method for proving inductive theorems. Indeed in this kind of theory it is possible to affirm that a theorem T is a consequence of a set of axioms A, w.r.t. replacement of equals by equals and induction iff T ⊕ A is consistent. This kind of theory are sometimes called maximally consistent or Hilbert-Post complete. So the proof is decomposed into two parts, a proof of relative completeness which is usually made once for all, and a proof of relative consistency. Methods for checking relative completeness are presented in the next section. The proof of consistency is based on the completion procedure, which generates rules from equations. If these equations contain defined functions they are proved by induction

as lemmas. Otherwise they contain only constructors and they are proved equationally by the means of the inductive completeness. Notice that methods based on or related to quasi–reducibility ‹PUE,84›, ‹KOU,85›, could be used to prove equations valid in T(C,CA).

2.4 – Fundamental Lemma

Now we may state the lemma which is the basis of the method of proof by consistency ‹REM, 82›, ‹PUE,83›, ‹PUE,84›, ‹KIR–84›. Let E be a set of equations the validity of which has to be proved in the initial algebra T(F,A).

Lemma:
If the specification (S,F,A) is complete w.r.t. (S,C,CA), then the following assertions are equivalent:
(i) (S,F,AUE) is consistent w.r.t. (S,C,CA).
(ii) (S,F,A) is consistent w.r.t. (S,C,CA) and every equation in E is valid in T(F,A).

Proof:
This proof is classical, but it can make things clear to give it.
* (i) ==› (ii)
Let u and v be two terms in T(C), with $u =_A v$. We have $u =_A v$ ==› $u =_{AUE} v$ ==› $u =_{CA} v$ by (i) hence (S,F,A) is consistent w.r.t. (S,C,CA). Suppose there is an equation u = v in E that is not valid in T(F,A) therefore there is a ground substitution σ s.t. $\sigma(u) \neq_A \sigma(v)$. Since σ(u) and σ(v) are two ground terms of T(F) and (S,F,A) is complete w.r.t. (S,C,CA), there are two terms u_0 and v_0 in T(C) s.t. $\sigma(u) =_A u_0$ and $\sigma(v) =_A v_0$. $\sigma(u) \neq_A \sigma(v)$ ==› $u_0 \neq_A v_0$ ==› $u_0 \neq_{CA} v_0$ this last implication comes from the fact that CA is a subtheory of A. On the other hand $u =_E v$ ==› $\sigma(u) =_{AUE} \sigma(v)$ and (i) says that $\sigma(u) =_{AUE} \sigma(v)$ ==› $u_0 =_{CA} v_0$ this is a contradiction.
* (ii) ==› (i)
Let u and v be two terms in T(C) $u =_{AUE} v$ ==› $u =_A v$ because every equation in E is valid in T(F,A). On the other hand $u =_A v$ ==› $u =_{CA} v$. because (S,F,A) is consistent w.r.t. (S,C,CA). Therefore (S,F,AUE) is consistent w.r.t. (S,C,CA).

The intuitive meaning of this lemma is the following: in order to prove a theorem in the initial algebra of a complete specification, the theorem is added to the specification and the consistency of this new specification is checked. If it is still consistent the theorem is valid, otherwise it is not. This method relies heavily upon the check of completeness of the specification (S,F,A) w.r.t. (S,C,CA). If this is done, the key of the method is now to prove the consistency of the specification w.r.t. to the theory of the constructors (S,C,CA), in other words we have to prove that E does not induce relations between terms of T(C,X) that were not already valid in T(C,CA).

2.5 – Completion procedure as a tool for proving consistency

The Knuth-Bendix completion procedure can be used to prove the relative consistency of a specification in the following way. Suppose (S, F, A) is a specification and we want to check its consistency w.r.t. (S, C, A). R is a convergent set of rules deduced form A and CR is a convergent set of rules deduced from CA. If (S, F, A) is not consistent, there exist two ground terms u and v such that $u =_A v$ and $u \neq_{CA} v$. We may suppose u and v in normal form w.r.t. CR. Notice that u and v are two different terms and at least one, say u, is not in normal form w.r.t R. This means there exists a rule $l \rightarrow r$ in R with $l \in T(C,X)$. On the other hand, if such a rule exists, this cannot correspond to an inductive property of (S, C, CA) indeed since (S, C, CA) is inductively complete, no rules wich is not already in (S, C, CA) can be added. Then the specification is not consistent. So to check the consistency, one runs the Knuth-Bendix completion procedure on the specification and one looks at whether rules with left-hand side in T(C,X) are introduced.

Thus, we suppose that

1. CA has been compiled by the completion procedure, into a convergent i.e., terminating and confluent rewriting system CR,

2. CR and DA have been also compiled by the completion procedure, into the rewriting system CR ⊕ DR.

We then run the completion procedure initialized by the set of equations E and the set of rewrite rules CR ⊕ DR. In this process, we assume $>>$ is a well-founded partial ordering on terms, compatible with the term structure and stable by substitution, with which we prove the termination of the successive sets of rewriting rules. In some cases, this ordering can be defined incrementaly ‹F-D,85›, ‹LES, 84›.

The only difference between a classical completion procedure and a procedure that checks consistency (see Appendix 1 for its complete text) occurs in the step in which a pair of terms, produced by the procedure i.e. coming from either a simplified critical pair or a reduced rewrite rule, is considered for orientation before being added as a new rewrite rule. In order to check if no new relation among constructors is introduced, this step is modified as follows, assuming that (u,v) is the current candidate as a rewrite rule, with $u \neq v$:

- if u and v are in T(C,X) then signal "DISPROOF".
- if u is in T(C,X) and v is not in T(C,X) then
  - if $v >> u$ then introduce the new rule $v \rightarrow u$.
  - otherwise signal "FAILURE".
- if v is in T(C,X) and u is not in T(C,X) then

- if u >> v then introduce the new rule u → v.
- otherwise signal "FAILURE".

– if u and v are not in T(C,X) then
- if v >> u then introduce the new rule v → u.
- if u >> v then introduce the new rule u → v.

– otherwise signal "FAILURE".

Theorem:
Let CA satisfy the inductive completeness;
– if the procedure stops with success or if it runs forever and the hypothesis of fairness in the choice of the rules is fulfilled then (S,F,AUE) is consistent w.r.t. (S,C,CA).
– if the procedure signals DISPROOF then (S,F,AUE) is inconsistent w.r.t. (S,C,CA).
– if the procedure signals FAILURE then nothing can be concluded.

Proof:
Let Q be the final rewriting system constructed by the procedure. Q may be finite or infinite, Q is of the form CR ⊕ DQ and no left–hand side of rules in DQ is in T(C,X). Moreover CR is kept without any modification all along the run of the procedure.

1. The procedure returns a finite rewriting system Q or runs forever.
   Let t and t'be two terms of T(C) $t =_{AUE} t' \Longrightarrow t =_{Q} t'$since $=_{Q}$ is equivalent to $=_{AUE}$ and on the other hand $t =_{Q} t' \Longrightarrow t =_{CR} t'$by the form of the rules in Q. The last equality is also $t =_{CA} t'$ therefore (S,F,AUE) is consistent w.r.t. (S,C,CA).

2. The procedure signals DISPROOF.
   We know all the equations produced by the completion procedure are consequences of AUE. When it stops the procedure has created a pair $(u,v) \in T(C,X) \times T(C,X)$ s.t. $u =_{AUE} v$ but $u \neq v$. Since u,v are normalized and belong to T(C,X), it means that $u \neq_{CR} v$. Since CR is inductively complete, it means that $u \neq_{ind(CR)} v$. This means there exists a ground instance σ, s.t. $\sigma(u) \neq_{CR} \sigma(v)$ and obviously $\sigma(u) =_{AUE} \sigma(v)$ which shows the inconsistency of (S,F,AUE) w.r.t. (S,C,CA).

Remark 1:
The first version of the previous algorithm and the ideas for preserving the set of irreducible ground instances of a term rewriting system can be tracked back in Remy <REM,82>, in Srivas <SRI,82> and in Dershowitz <DER,83>.

Remark 2:
Usually the orderings >> used for proving termination do not make a term in T(C,X) greater than a term in T(F,X)–T(C,X) so the first two FAILURE conditions

will never occur, in this case.

Remark 3:
It is possible to work with non inductively complete theories for (S, C, CA) as indicated by L. Puel ‹PUE,83›. Indeed let u and v be two different terms in T(C,X), if there are in T(C) then the procedure signals *disproof*, otherwise it has to prove them by induction. To do that, L. Puel suggests to consider the variables that occur in u and v as constant, to add the rule $u \rightarrow v$ to CR and to use it to prove all the derivated equations. A derivated equation is an equation where a variable is replaced by a term of the form $c(x_1,\ldots,x_n)$ for a constructor c. If this fails it is possible to iterate the process. For instance, if u(x) = v(x) is such an equation in the integers, one tries to prove u(0) = v(0) and u(s(x)) = v(s(x)), after adding u(x) = v(x) to CR. If this fails one tries to prove u(s(0)) = v(s(0)) and u(s(s(x))) = v(s(s(x))) after adding u(s(x)) = v(s(x)) to CR. In some sense, in this specific case, a classical proof by induction replace the proof by consistency as indicated by Boyer and Moore in ‹B–M,79›.

3 – COMPLETENESS OF A SPECIFICATION

In this section we suppose that the specification A is associated with a convergent term rewriting system R produced from A by the completion procedure. We use the notation (S,F,R) instead of (S,F,A) and we use this system R to check the completeness of A.

Obviously (S,F,R) is complete w.r.t. (S,C,CR) if the R–normal form of any ground term of T(F) belongs to T(C). This will be checked using the following concepts:

Definition: C–completeness
Let f be a defined operator and n its arity. f is C–complete w.r.t. (S,F,R) iff for all normal forms $u_1,\ldots,u_n$ in T(C), $f(u_1,\ldots,u_n)$ is R–reducible.

Notations:
Let us take the following notations, where f is a defined operator.
TC(f,X) = the set of f–rooted terms that are terms with root f and arguments in T(C,X).
TC(f) = the set of terms with root f and arguments in T(C).
Let t be a term of TC(f,X) and T a subset of TC(f,X).
Var(t) = the set of variables which have at least one occurrence in t.
G(t) = the set of ground terms that are instance of t.
G(T) = $\cup_{t\in T}$ G(t).
L(f,R) = the set of left–hand sides of rules of R s.t. their root is f.
$x_1,\ldots,x_n$ are variables that are supposed distinct.

We have:

Lemma:
If each defined operator in D is C–complete, then the specification (S,F,R) is complete w.r.t. (S,C,CR).

Proof:
If (S,F,R) is not complete, there exists a term t of T(F) s.t. its R–normal form t_0 does not belong to T(C). Hence there is a subterm $f(u_1,\ldots,u_n)$ of t_0 with $f \in D$ and $u_1,\ldots,u_n \in T(C)$; and $f(u_1,\ldots,u_n)$ is not R–reducible which is in contradiction with the definition of C–completeness of f.

Theorem:
f is C–complete if all terms of $G(f(x_1,\ldots,x_n)) - G(L(f,R))$ are CR–reducible.

Proof:
$G(f(x_1,\ldots,x_n))$ is the set of all f–rooted ground terms. If every term that is not in G(L(f,R)) i.e., not reducible by a f–rooted rule, is CR–reducible then all f–rooted

ground terms are R–reducible and f is C–complete.

Example:
Let C={ 0 , opp , succ }, D={ + } and suppose the set of axioms is:

opp(0) → 0
opp(opp(x)) → x
succ(opp(succ(x))) → opp(x)
0 + x → x
succ(x) + y → succ(x + y)
opp(succ(x)) + y → opp(succ(x + opp(y)))

+ is C–complete, indeed:
$G(x_1 + x_2) - G(L(+,R))$ = { t / t is an instance of opp(opp(x)) + y or opp(0) + y } contains only CR–reducible terms.

Remark:
If there are relations between constructors i.e., CR is not empty, the theorem gives only a sufficient condition for testing C–completeness. A way to strengthen this condition is to require CR–inductive–reducibility (or CR–quasi–reducibility ‹KOU,85›) which is the reducibility of all ground instances, instead of CR–reducibility. Practically, the method to prove the C–completeness is to prove that each term of $G(f(x_1,\ldots,x_n))$ is either an instance of a left–hand side of a rule or is CR–reducible.

In order to use this theorem in practice we introduce the following concepts:

Definition: cover
Let M and N two subsets of T(F,X). We say that M covers N iff G(N) is a subset of G(M), i.e., each instance of a term in N is an instance of a term in M.

Example:
{ S(x) + y ; P(x) + y } does not cover { x + y } but
{ 0 + y ; S(x) + y ; P(x) + y } covers { x + y } .

The goal of the *cover* is to apply the theorem by finding the part K of $G(f(x_1,\ldots,x_n))$ which is not covered by G(L(f,R)) and by checking that the terms of K are CR–reducible. In practice, we cannot use this technique directly because it would induce manipulations of infinite sets of terms, so we introduce the following concept:

Definition: *complement of a term*
Let t be a term of T(C,X), we call complement of t, any finite set C(t) of terms s.t. $T(C) = G(t) \oplus G(C(t))$.
Especially if $t \in X$, C(t) is empty, since G(t) = T(C).

The following proposition gives a constructive definition of a complement of a *linear term* in T(C,X) which is a term with at most one occurrence of each variable.

Proposition:
If t is a linear term of T(C,X) s.t. $t = c_j(t_1,\ldots,t_{nj})$, $C = \{ c_1,\ldots,c_m \}$ where nh is the arity of c_h and

$$C(t) = \{ c_i(x_1,\ldots,x_{ni}) \mid x_1,\ldots,x_{ni} \in X \ \&\ 1 \leq i \leq m \ \&\ i \neq j \}$$
$$\cup_{1 \leq k \leq nj} \{ c_j(t_1,\ldots,t_{k-1}, v, x_{k+1},\ldots,x_{nj}) \mid x_{k+1},\ldots,x_{nj} \in X \ \&\ v \in C(t_k) \}.$$

then C(t) is a complement of t.
N.B. The variables in $t_1,\ldots,t_{k-1}, v, x_{k+1},\ldots,x_{nj}$ are supposed different.

Proof:
* $G(t) \cap G(C(t)) = \emptyset$ is obvious.
* $G(t) \cup G(C(t)) = T(C)$ indeed:
let u be a term of T(C) and $t = c_j(t_1,\ldots,t_{nj})$;
if the root of u is c_i with $i \neq j$
then $u \in G(C(t))$ because u is an instance of $c_i(x_1,\ldots,x_{ni})$.
else let $u = c_j(u_1,\ldots,u_{nj})$
if there exists σ s.t. $\sigma(t) = u$ then $u \in G(t)$
else we build σ in the following way,
let $k \in [1..nj]$ s.t.$(\forall h < k)$ $(\exists \sigma_h)$ $\sigma_h(t_h) = u_h$
and there is no substitution that matches t_k with u_k,
therefore there exists σ_k and $v \in C(t_k)$ with $\sigma_k(v) = u_k$. Since
the term t is linear it is possible to define the substitution σ as follows:
case x occurs in t_h with $h < k$ $\quad \sigma(x) = \sigma_h(x)$
x occurs in v $\quad \sigma(x) = \sigma_k(x)$
$x = x_h$ with $h > k$ $\quad \sigma(x) = u_h$.
It is now obvious that $\sigma(c_j(t_1,\ldots,t_{k-1}, v, x_{k+1},\ldots,x_{nj})) = u$ and $u \in G(C(t))$.

Example:
C(0)={ S(x) ; P(x) }.
C(S(0))={ 0 ; P(x) ; S(S(x)) ; S(P(x)) }.

Definition: *linear substitution*
A substitution σ is linear if it satisfies these equivalent properties:
(i) $(\forall x \in D(\sigma))$ $\sigma(x)$ is linear and $(\forall y \in D(\sigma))$ $x \neq y \Rightarrow Var(\sigma(x)) \cap Var(\sigma(y)) = \emptyset$.
(ii) σ transforms any linear term into a linear term.

Definition: *complement of a substitution*
Let σ be a linear substitution in T(C,X) and $D(\sigma)$ its domain, we call complement of σ the set $C(\sigma)$ of all substitutions ρ such that:
(i) $\rho \neq \sigma$.

(ii) $D(\rho) = D(\sigma)$.
(iii) $(\forall x \in D(\rho))\ \rho(x) \in C(\sigma(x))$ or $\rho(x) = \sigma(x)$.

Example:
Let $\sigma = \{ x \leftarrow 0 ;\ y \leftarrow S(v) ;\ z \leftarrow u \}$
$C(\sigma) = \{ \{x \leftarrow S(x_1) , y \leftarrow S(v) , z \leftarrow u\} ; \{x \leftarrow P(x_1) , y \leftarrow S(v) , z \leftarrow u\} ;$
$\{x \leftarrow P(x_1) , y \leftarrow 0 , z \leftarrow u\} ; \{x \leftarrow 0 , y \leftarrow 0 , z \leftarrow u\} ;$
$\{x \leftarrow 0 , y \leftarrow P(x_2) , z \leftarrow u\} ; \{x \leftarrow S(x_1) , y \leftarrow 0 , z \leftarrow u\} ;$
$\{x \leftarrow P(x_1) , y \leftarrow P(x_2) , z \leftarrow u\} ; \{x \leftarrow S(x_1) , y \leftarrow P(x_2) , z \leftarrow u\} \}$.

The complement of a substitution is a nice way to compute a basis for the set $G(t)-G(\sigma(t))$ for $t \in T(C,X)$ and it is an easy task once one knows how to compute the complement of a term in T(C,X), after the previous proposition.

Proposition:
Let t be a term in TC(f,X) and σ a linear substitution
$G(t) = \cup_{\rho \in C(\sigma)} G(\rho(t)) \cup G(\sigma(t))$ and

$(\forall \rho \in C(\sigma))\ G(\rho(t)) \cap G(\sigma(t)) = \emptyset$ or $\rho(t) = \sigma(t)$.

Proof:
By definition of the complement of a substitution.

Note that the restriction is about the linearity of the substitution not about the linearity of t or $\sigma(t)$. For example, if $C=\{0,S\}$ and $\sigma=\{y \leftarrow S(x)\}$, then
$G(f(y,y)) - G(\sigma(f(y,y))) = G(f(y,y)) - G(f(S(x),S(x))) = G(f(0,0))$.
This can be computed althought f(y,y) and $\sigma(f(y,y))$ are not linear. The following theorem gives a sufficient condition for testing the concept of cover:

Theorem:
M covers N if one of the following conditions are satisfied:

(i) N is empty.

(ii) There exists two terms $m \in M$ and $n \in N$ s.t. m and n are unified by a linear subsitution σ and $M-\{m\} \cup (\{\rho(m) \mid \rho \in C(\sigma)\}-\{\sigma(m)\})$ covers $N-\{n\} \cup (\{\rho(n) \mid \rho \in C(\sigma)\}-\{\sigma(n)\})$.

Proof:
Let $M' = M-\{m\} \cup (\{\rho(m) \mid \rho \in C(\sigma)\}-\{m\})$ and $N' = N-\{n\} \cup (\{\rho(n) \mid \rho \in C(\sigma)\}-\{n\})$.
By the previous proposition:

$$G(M) = G(M-\{m\}) \oplus G(m) = G(M-\{m\}) \oplus (\cup_{\rho \in C(\sigma)} G(\rho(m)) \cup G(\sigma(m)))$$
$$= G(M') \oplus G(\sigma(m))$$

similarly,

$$G(N) = G(N-\{n\}) \oplus G(n) = G(N-\{n\}) \oplus (\cup_{\rho \in C(\sigma)} G(\rho(n)) \cup G(\sigma(n))).$$
$$= G(N') \oplus G(\sigma(n))$$

Thus, since $\sigma(m) = \sigma(n)$, G(M) contains G(N) if and only G(M') contains G(N').

Now we may state the theorem for testing the completeness of a specification w.r.t. to a specification of constructors ‹THI,84›. Let us start with $M_0=\{L(f,R)\}$ and $N_0=\{f(x_1,\ldots,x_n)\}$, and repeat the operations described in the part (ii) of the theorem until N is empty or no $m \in M$ and no $n \in N$ can be unified by a linear substitution. This will eventually happen since it can be proven that terms of the form $f(t_1,\ldots,t_n)$ produced by unification and computation of complements have no t_i with a size greater than the size of the same terms in the data M_0 and N_0. Let us call M_f and N_f the final results.

Theorem:
- if M_f and N_f are empty, f is C–complete without ambiguity.
- if M_f is not empty and N_f is empty, f is C–complete but all terms in M_f are defined more than once.
- if N_f is not empty, f may be not C–complete because the terms in N_f are not defined.

The third kind of result returned by the algorithm is really interesting, since it gives the patterns where the function has to be defined. This feature makes our algorithm really handy in a environment for functional programming or abstract data types, as seen on the examples in Appendix 3, since these patterns are obtained by unification they are in some sense the most general ones and operationally they seem to be better than methods based on unfolding a tree as proposed by E. Kounalis ‹KOU,85›, D. Plaisted ‹PLA,85› and D. Kapur, P. Narendran and Zhang ‹KNZ,86›. The concept of coverture appeard in ‹THI,84› but was presented independently in a recent paper by Lassez and Marriott in the context of programming logic and automated learning ‹L–M,87›.

Remark:
Note that if M contains only linear terms this procedure is a decision procedure for C–completeness ‹N–W,83› but it is worthwile to emphasize that this method can handle non linear term rewriting systems. In general, one takes $N=\{ f(x_1,\ldots,x_n) \}$, but the algorithm may fail because all the substitutions that unify the terms are not linear, usually another complete and non ambiguous set N could lead to a success. This is the case in the following example due to E. Kounalis ‹KOU,85›.

Example:
This function computes the digit to *carry* in an binary additioner. It is also the function *majority* in a fault tolerant system, see also ‹TOY,86›.
Let C={ 1 , 0 } , D={ f } and the set of axioms:

$f(x,x,y) \rightarrow x$
$f(x,y,x) \rightarrow x$
$f(y,x,x) \rightarrow x$

To test the completeness of this specification we can take $N=\{ f(u,v,1) , f(u,v,0) \}$ indeed:

$\{f(x,x,y),f(x,y,x),f(y,x,x)\}$ covers $\{f(u,v,1),f(u,v,0)\}$ because
$f(x,x,y)$ and $f(u,v,1)$ are unified by $\sigma=\{u\leftarrow x,v\leftarrow x,y\leftarrow 1\}$ which is not linear.
However
$f(x,y,x)$ and $f(u,v,1)$ are unified by $\sigma=\{x\leftarrow 1,y\leftarrow v,u\leftarrow 1\}$ which is not linear.
$C(\sigma)=\{\{x\leftarrow 0,y\leftarrow v,u\leftarrow 1\},\{x\leftarrow 1,y\leftarrow v,u\leftarrow 0\},$
$\{x\leftarrow 0,y\leftarrow v,u\leftarrow 0\}\}$

$\{f(x,x,y),f(y,x,x),f(0,v,0)\}$ covers $\{f(u,v,0),f(0,v,1)\}$ because
$f(y,x,x)$ and $f(u,v,0)$ are unified by $\sigma=\{x\leftarrow u,y\leftarrow 0,v\leftarrow 0\}$
$C(\sigma)=\{\{x\leftarrow u,y\leftarrow 0,v\leftarrow 1\},\{x\leftarrow u,y\leftarrow 1,v\leftarrow 0\},$
$\{x\leftarrow u,y\leftarrow 1,v\leftarrow 1\}\}$

$\{f(x,x,y),f(0,v,0),f(u,1,1)\}$ covers $\{f(0,v,1),f(u,1,0)\}$ because
$f(x,x,y)$ and $f(0,v,1)$ are unified by $\sigma=\{x\leftarrow 0,v\leftarrow 0,y\leftarrow 1\}$
$C(\sigma)=\{\{x\leftarrow 1,v\leftarrow 0,y\leftarrow 1\},\{x\leftarrow 0,v\leftarrow 1,y\leftarrow 0\},$
$\{x\leftarrow 1,v\leftarrow 1,y\leftarrow 1\},\{x\leftarrow 1,v\leftarrow 0,y\leftarrow 0\},$
$\{x\leftarrow 1,v\leftarrow 1,y\leftarrow 0\},\{x\leftarrow 0,v\leftarrow 1,y\leftarrow 1\},$
$\{x\leftarrow 0,v\leftarrow 0,y\leftarrow 0\}\}$

$\{f(0,v,0),f(u,1,1),f(1,1,1),f(1,1,0),f(0,0,0)\}$ covers $\{f(u,1,0),f(0,1,1)\}$ because
$f(0,v,0)$ and $f(u,1,0)$ are unified by $\sigma=\{v\leftarrow 1,u\leftarrow 0\}$
$C(\sigma)=\{\{v\leftarrow 0,u\leftarrow 0\},\{v\leftarrow 0,u\leftarrow 1\},\{v\leftarrow 1,u\leftarrow 1\}\}$

$\{f(u,1,1),f(1,1,1),f(1,1,0),f(0,0,0)\}$ covers $\{f(1,1,0),f(0,1,1)\}$ because
$f(u,1,1)$ and $f(0,1,1)$ are unified by $\sigma=\{u\leftarrow 0\}$ and $C(\sigma)=\{u\leftarrow 1\}$

$\{f(1,1,1),f(1,1,0),f(0,0,0)\}$ covers $\{f(1,1,0)\}$ because
$f(1,1,0)$ and $f(1,1.0)$ are unified by the substitution identity

$\{f(1,1,1),f(0,0,0)\}$ covers $\emptyset$
hence f is C-complete and the two terms $f(1,1,1)$ and $f(0,0,0)$
are defined more than one.

4 – THE IMPLEMENTATION IN REVE

To make the features presented in this paper available in REVE ‹FOR,84›, we have implemented new commands or changed the existing ones.

constructor. This command declares the constructors of the specification. As previously, it sets the precedence such that the constructors are less than any other operator.

complete. This command checks whether a specification is relatively complete, using the algorithm described in this paper. If it is not relatively complete, REVE tells the user where it could be defined. If REVE fails because it cannot find a linear substitution, it allows the user to use a reference which is not $f(x_1,\ldots,x_n)$.

prove. The format of this command is not changed w.r.t. the previous versions of REVE. The semantics is different, since it accepts relations among the constructors. It supposes that the relations between the constructors are inductively complete and the definitions of the operators are relatively complete. The first check has to be done by the user unless he takes one of the specifications quoted in this paper and the second one can be made by REVE.

As usually the consistency is tested by running the Knuth–Bendix completion algorithm.

5 – CONCLUSION

We have surveyed the method for proving inductive properties of functional programming, as actually implemented in REVE as shown on the examples run on a VAX at Nancy (Appendix 3). The main new algorithm is an algorithm for proving relative or sufficient completeness. Recent works by H. Comon ‹COM,86› and also by C. Kirchner and P. Lescanne ‹K–L,87› have shown how it could be extended.

We are fully aware that our current implementation does not implement the state of the art, but it is a step towards an implementation of more elaborated tools for proving inductives theorems. We are indeed currently pushing our works into several directions, for handling equational rewriting in REVE 3 ‹K–K,85›. for using the full power of J.P. Jouannaud and E. Kounalis approach ‹J–K,86› comparisons with the close methode proposed by D. Kapur, P. Narendran and H. Zhang will also be done ‹KNZ,85›. We are also looking at L. Puel's solution for dealing with non inductively complete theories and at L. Fribourg ‹FRI,86› suggestions for reducing the superpositions that are computed by the induction proofs. On another hand we are studying algorithmic solutions for testing inductive completeness.

REFERENCES

‹B-M,79› R.S. BOYER & J.S. MOORE, "A Computational Logic", Academic Press, (1979).

‹BMS,80› R.M. BURSTALL, D.B. MACQUEEN, D.T. SANNELLA, "HOPE : An experimental applicative language", Conference Record of the 1980, LISP conference, California, (1980), pp. 136–143.

‹COM,86› H. COMON, "Sufficient Completeness, Term Rewriting Systems and Anti-Unification", 8th International Conference on Automated Deduction, Lecture Notes in Computer Science, vol. 230, Springer Verlag, Oxford England, (1986), pp. 128–140.

‹CPH,85› G. COUSINEAU, L. PAULSON, G. HUET, R. MILNER, M. GORDON, C. WADSWORTH, "The ML Handbook", INRIA, Rocquencourt, France, (1985).

‹DER,83› N. DERSHOWITZ, "Well-Founded Orderings", ATR-83(8478)-3, Information Science Research Office, The Aerospace Corporation, El Segundo, California USA, (1983).

‹F-D,85› R. FORGAARD & D. DETLEFS, "An incremental algorithm for proving termination of term rewriting systems", Proc. 1rst International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 202, Springer Verlag, (1985), pp. 255–270.

‹F-G,83› R. FORGAARD & J.V. GUTTAG, "REVE : a term rewriting system generator with failure-resistant", Laboratory for computer science MIT (1983).

‹FAG,84› F. FAGES, "Le système KB, manuel de référence, présentation et bibliographie, mise en oeuvre" R.G. 10.84, GRECO de Programmation Bordeaux, (1984).

‹FGJM,84› K. FUTATSUGI & J.A. GOGUEN & J.P. JOUANNAUD & J. MESEGUER, "Principles of OBJ2", Proc. 12th ACM Principles Of Programming Languages Conference, (1984).

‹FOR,84› R. FORGAARD, "A program for generating and analyzing term rewriting system", Master of science in computer science MIT (1984).

‹FRI,86› L. FRIBOURG, "A strong restriction of the completion procedure for theories with constructors" International Conference on Automata Languages and Programming, Lecture Notes in Computer Science, vol. 226, Springer Verlag, Rennes France, (1986), pp. 105–115.

‹G–H,78› J.V. GUTTAG & J.J. HORNING, "The algebraic specification of abstract data types", Acta Informatica, vol. 10, (1978), pp. 27–52.

‹GOG,80› J.A. GOGUEN, "How to prove algebraic inductive hypotheses with application to the correctness of data types implementation", proc. 5th International Conference on Automated Deduction, Lecture Notes in Computer Science, vol. 87, Springer Verlag, les Arcs, (1980), pp. 356–373.

‹H–H,82› G. HUET & J.M. HULLOT, "Proofs By Induction in Equational Theories With Constructors", J. Comp. Sys. Sc., vol. 25, (1982), pp. 239–266.

‹HEE,85› J. HEERING, "Partial evaluation and ω–completeness of algebraic specifications", Theoretical Computer Science, vol. 43, (1986), pp. 149–167.

‹HEN,77› L. HENKIN, "The logic of equality", The American Mathematical Monthly, vol. 84, (1977), pp. 597–612.

‹J–K,86› J.P. JOUANNAUD & E. KOUNALIS, "Proof by Induction in Equational Theories without Constructors" Proc. 1rst IEEE Symp. on Logic in Computer Science, Cambridge, Massachussetts, USA (1986).

‹K–K,85› C. KIRCHNER & H. KIRCHNER, "Implementation of a general completion procedure parameterized by built–in theories and strategies", Proceedings of the EUROCAL 85 conference, Lecture Notes in Computer Science, vol. 2, Springer Verlag, (1985).

‹K–L,87› C. KIRCHNER & P. LESCANNE, "Solving Disequations", Proc. of the 2nd IEEE Symp. on Logic in Computer Science, (1987).

‹K–M,87› J.–L. LASSEZ & K. MARRIOTT, "Explicit Representation of Terms Defined by Counter Examples", Journal of Automated Reasoning 3, (1987).

‹KNZ,85› D. KAPUR, P. NARENDRAN & H. ZHANG, "On Sufficient Completeness and Related Properties of Term Rewriting Systems", Unpublished Manuscipt, General Electric R&D Center, Schenectady, Submitted to Acta Informatica, (1985).

‹KNZ,86› D. KAPUR, P. NARENDRAN & H. ZHANG, "Proof by induction using test sets", 8th International Conference on Automated Deduction, Lecture Notes in Computer Science, vol. 230, Springer Verlag, Oxford England, (1986), pp. 99–117.

‹KIR,84› H. KIRCHNER, "A general inductive completion algorithm and application to abstract data types", Proc. 7th international Conference on Automated Deduction, Napa Valley California, USA, Lecture Notes in Computer

Science, vol. 170, Springer Verlag, (1984), pp 282-302.

‹KIR,85› H. KIRCHNER, "Preuves par complétion dans les variétés d'algèbres", Thèse d'Etat Université Nancy 1, CRIN, NANCY, (1985).

‹KOU,85› E. KOUNALIS, "Validation de spécifications algébriques par complétion inductive", Thèse de doctorat Université Nancy 1, CRIN, NANCY, (1985).

‹LAN,81› D.S. LANKFORD, "A simple explanation of inductionless induction", Louisiana Tech. Univ., dep. of math., Memo MTP-14, Ruston, (1981).

‹LES,83› P. LESCANNE, "Computer Experiments with the REVE Term Rewriting System Generator" 10th ACM Principles Of Programming Languages, Austin, Texas, (1983), pp. 99-108.

‹LES,84› P. LESCANNE, "Uniform termination of term rewriting systems - Recursive decomposition ordering with status", Proceeding 9th colloquium on trees in Algebra and Programming, BORDEAUX, Cambridge U. Press, (1984), pp. 182-194.

‹M-G,83› J. MESEGUER & J.A. GOGUEN, "Initiality, induction, and computability", CSL Technical Report 140, SRI California (1983).

‹M-K,84› D.R. MUSSER and D. KAPUR, "Proof by consistency", Proc. of an NSF, Workshop N.84 GEN008, April 1984.

‹MIL,84› R. MILNER, "A proposal for standard ML", proc. ACM Conference of LISP and Functional Programming, Austin (1984).

‹MUS,80› D.R. MUSSER, "On proving inductive properties of abstract data types", proc. 7th ACM Principles Of Programming Languages Conference, Las Vegas (1980), pp. 154-162.

‹N-W,83› T. NIPKOW & G. WEIKUM, "A decidability result about sufficient completeness of axiomatically specified abstract data types", 6th GI conference Dortmund, Lecture Notes in Computer Science, vol. 145, Springer Verlag, (1983), pp. 257-268.

‹PAU,84› E. PAUL, "Proof by induction in equational theories with relations between constructors", Proceeding 9th colloquium on trees in Algebra and Programming, BORDEAUX, Cambridge U. Press, (1984), pp. 210-225.

‹PLA,85› D. PLAISTED, "Semantic confluence tests and Completion Methods" Journal Information and Control, vol. 65, (1985), pp. 182-215.

‹PLO,74› G.D. PLOTKIN, "The $\lambda\kappa\beta\eta$–calculus is ω–incomplete", Journal Symbolic Logic, vol. 39, (1974), pp. 313–317.

‹PUE,83› L. PUEL, "Preuve dans l'algèbre terminale, Algorithmes de complétion." Thèse de troisième cycle, Université Paris VII, (1983).

‹PUE,84› L. PUEL, "Proof in the Final Algebra" Proc. 9th colloquium on trees in Algebra and Programming, BORDEAUX, Cambridge U. Press, (1984), pp. 227–242.

‹REM,82› J.L. REMY, "Etudes des systèmes de réécriture conditionnelle et applications aux types abstraits algébriques", Thèse d'Etat, INPL, Nancy, (1982).

‹SRI,82› M.K. SRIVAS, "Automatic synthesis of implementations for abstract data types from algebraic specifications", MIT–LCS–TR.276, (1982).

‹TAY,79› W. TAYLOR, "Equational Logic", Houston Journal of Mathematics, Survey, (1979); see also in G. GRATZER "Universal Algebra" (Second Edition), Springer Verlag, (1979), Appendix 4.

‹THI,84› J.J. THIEL, "Stop losing sleep over incomplete data type specifications", Proc. 11th ACM confeence on Principles of Programming Languages, Salt Lake City Utah, USA,(1984).

‹TOY,86› Y. TOYAMA, "Counterexamples to Terminating for the Direct Sum of Term Rewriting Systems", Submitted to Information Processing Letters, (1986).

‹TUR,85› D.A. TURNER, "Miranda: A non–strict functional language with polymorphic types", 2nd Conference on Functional Programmning and Computer Architecture, Lecture Notes in Computer Science, vol. 201, Springer Verlag, (1985), pp. 1–16.

APPENDIX 1

initialization : P = E ; R = CR ⊕ DR

completion (P,R,>>)
  if P is not empty *then* choose a pair (p,q) in P; p'← p↓; q' ← q↓
    if p' = q' *then return*(completion (P–{(p,q)},R,>>))
    elseif (p',q') ∈ T(C,X)xT(C,X) *then signal* DISPROOF
    elseif p' >> q' *then* l ← q' & r ← p'
    elseif q' >> p' *then* l ← p' & r ← q'
      else signal FAILURE
    (P,R) = simplification (P–{(p,q)},R,l → r)
    R = completion (P,RU{l →r},>>)
  elseif all rules in R are marked *then return*(R)
  else choose an unmarked rule l → r
    (P,R) = critical-pairs (l → r,R)
    mark the rule l → r in R
    return(completion (P,R,>>))
end.

p↓ : the normal form of the term p.
simplification : simplification of other rules and equations using the new added rules. critical-pairs : computation of critical pairs between rules.

APPENDIX 2

We give here some examples of inductively complete specification and their proofs. By definition , a specification (S,F,A) is inductively complete iff for every terms u and v in T(F,X):

(1) $u =_{ind(A)} v \Longrightarrow u =_A v$.

Let u↓ denote the R–normal form of u, where R is a convergent rewriting system associated with A. Obviously,

$u =_{ind(A)} v \Longleftrightarrow u\downarrow =_{ind(A)} v\downarrow$,

and

$u =_A v \Longleftrightarrow u\downarrow = v\downarrow$

i.e., u↓ is syntactically the same term as v↓. Thus (1) is equivalent to

(2) $u\downarrow =_{ind(A)} v\downarrow \Longrightarrow u\downarrow = v\downarrow$.

In other words

(for all ground substitution σ, $\sigma(u\downarrow)\downarrow = \sigma(v\downarrow)\downarrow) \Longrightarrow u\downarrow = v\downarrow$.

By contraposition the inductive completeness can be expressed as

$u\downarrow \neq v\downarrow \Longrightarrow$ (there exists a σ ground substitution s.t. $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$).

Example 1:

Let F={ 0,opp,succ } and suppose the set of axioms is:

opp(0) → 0

opp(opp(x)) → x

succ(opp(succ(x))) → opp(x)

The normal forms of the ground terms are one of the following:

0

$succ^n(0)$ for n>0 (where $succ^n(0)$ is succ(succ(...(succ(0))...)) n times)

$opp(succ^n(0))$ for n>0

The normal forms of the terms of the free algebra T(F,{x},A) are one of the following:

$succ^n(x)$ for n>0

$succ^n(opp(x))$ for n≧0 (where $succ^0(opp(x))$ is opp(x))

$opp(succ^n(x))$ for n>0

$opp(succ^n(opp(x)))$ for n≧0

Let u↓ and v↓ be two terms of T(F,{x},A) s.t. $u\downarrow \neq v\downarrow$

case $u\downarrow = succ^n(x)$ then

case $v\downarrow = succ^m(opp(x))$ then

case n = m then let σ={ x ← succ(0) }

$\sigma(u\downarrow)\downarrow = succ^{n+1}(0)$ and $\sigma(v\downarrow)\downarrow = succ^{n-1}(0)$

thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$

$n \neq m$ then let σ={ x ← 0 }

$\sigma(u\downarrow)\downarrow = succ^n(0)$ and $\sigma(v\downarrow)\downarrow = succ^m(0)$

thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$

$v\downarrow = opp(succ^m(x))$ then let σ={ x ← 0 }

$\sigma(u\downarrow)\downarrow=succ^n(0)$ and $\sigma(v\downarrow)\downarrow= opp(succ^m(0))$
thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$

$v\downarrow = opp(succ^m(opp(x)))$ then let $\sigma=\{ x \leftarrow 0 \}$
$\sigma(u\downarrow)\downarrow = succ^n(0)$ and $\sigma(v\downarrow)\downarrow = opp(succ^m(0))$
thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$

$u\downarrow = succ^n(opp(x))$ then
case $v\downarrow = opp(succ^m(x))$ then let $\sigma=\{ x \leftarrow 0 \}$
$\sigma(u\downarrow)\downarrow = succ^n(0)$ and $\sigma(v\downarrow)\downarrow = opp(succ^m(0))$
thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$

$v\downarrow = opp(succ^m(opp(x)))$ then $\sigma=\{ x \leftarrow 0 \}$
$\sigma(u\downarrow)\downarrow = succ^n(0)$ and $\sigma(v\downarrow)\downarrow = opp(succ^m(0))$
thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$

$u\downarrow = opp(succ^n(x))$ then $v\downarrow = opp(succ^m(opp(x)))$
case $n = m$ then let $\sigma=\{ x \leftarrow succ(0) \}$
$\sigma(u\downarrow)\downarrow = opp(succ^{n+1}(0))$ and $\sigma(v\downarrow)\downarrow = opp(succ^{n-1}(0))$
thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$

$n \neq m$ then let $\sigma=\{ x \leftarrow 0 \}$
$\sigma(u\downarrow)\downarrow = opp(succ^n(0))$ and $\sigma(v\downarrow)\downarrow = opp(succ^m(0))$
thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$

Example 2:
Let $F=\{ 0 \cup \{ a,b\} , . \}$ and suppose the set of axioms is:

$x.0 \rightarrow x$
$0.x \rightarrow x$
$(x.y).z \rightarrow x.(y.z)$

the initial algebra T(F,A) is the free monoid $(\{a,b\})^*$ and the free algebra T(F,X,A) is equivalent to $(\{a,b\} \cup X)^*$. Let $u\downarrow$ and $v\downarrow$ two terms of T(F,X,A) s.t. $u\downarrow \neq v\downarrow$ therefore there exist $c,d \in \{a,b\} \cup X$ s.t. $c \neq d$, $u\downarrow = t.c.t'$ and $v\downarrow = t.d.t''$
case $c \in \{a,b\}$ and $d \in \{a,b\}$ then $\forall\sigma \quad \sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$
$c \in \{a,b\}$ and $d = x$ then if $c = a$ (resp. $c = b$) then let $\sigma=\{ x \leftarrow b$ (resp. $x \leftarrow a)\}$
thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$
$c = x$ and $d = y$ then let $\sigma=\{ x \leftarrow a , y \leftarrow b\}$
thus $\sigma(u\downarrow)\downarrow \neq \sigma(v\downarrow)\downarrow$

Example 3:
Let $F= \{0, 1, opp, +\}$ and suppose the set of axioms is

$0+x \rightarrow x$
$opp(0) \rightarrow 0$
$opp(opp(x)) \rightarrow x$
$opp(x+y) \rightarrow opp(x)+opp(y)$
$x+y = y+x$
$(x+y)+z = x+(y+z)$

The normal forms are $u = x_1+\ldots+x_m+y_1+\ldots+y_n$, with $y_j = opp(z_j)$. Let $u \neq u'$, then there exists either a variable, say x, that occurs in one side, say u, or a variable, say again x, that occurs in the first part of the normal form u and in the second part of the normal form u'. In both case define the substitution σ, s.t. $\sigma(x) = 1$ and $\sigma(y) = 0$ for $y \neq x$. In the first case, we get $\sigma(u) = 1$ and $\sigma(u') = 0$ and, in the second case, we get $\sigma(u) = 1$ and $\sigma(u') = opp(1)$.

APPENDIX 3

-> thaw list

No user equations.

No critical pair equations.

Rewrite rules:

1. (x @ null) -> x
2. (null @ x) -> x
3. flatten(null) -> null
4. flatten((unit(x) @ y)) -> (flatten(x) @ flatten(y))
5. flatten(unit(x)) -> flatten(x)
6. ((x @ y) @ z) -> (x @ (y @ z))

-> operators

Operator precedence:

flatten > { null, @, unit, ab, }
null(C) > { }
@(C) > { }
unit(C) > { }
a(C) > { }
b(C) > { }

-> complete

The constructors aren't free, I THINK THAT :

The operator(s):
flatten

is (are) not sufficiently complete.

I suggest to add rule(s) with following left_hand side(s):
flatten(a)

flatten(b)

flatten((a @ x1))

flatten((b @ x1))

–› add
Please type in additional equations, terminated with ‹ESC›:
flatten(a)==a
flatten(b)==b
flatten(a@x)==a@flatten(x)
flatten(b@x)==b@flatten(x)

Note that the following identifiers were parsed as nullary operators:
a b

–› complete

The system is not convergent; I run Knuth–Bendix to complete it.

...

All operators are sufficiently complete.

–› *This rewriting system can be used to prove:*
flatten(x@y) == flatten(x)@flatten(y)
flatten(flatten(x)) == flatten(x)
and disprove:
flatten(flatten(x)) == x

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique